Discrete logarithm

Application of discrete models

1 Modular exponentiation

From the previous document (Congruences), we know that congruences can be added, subtracted, multiplied (not always divided), and exponentiated. This document is about the latter operation. To repeat it once more:

Theorem 1.1. *If* $a \equiv b \pmod{m}$*, then* $\forall n \in \mathbb{N} : a^n \equiv b^n \pmod{m}$.

Example. $5 \equiv 12 \pmod{7} \implies 5^3 \equiv 12^3 \pmod{7}$, i.e. it doesn't matter which one of the congruent bases (5 and 12) are used for exponentiating.

(Warning: this is only true for the base, not for the exponent! For example, $5^{10} \not\equiv 5^3 \pmod{7}$. This will be discussed in the next document.)

In this document, we focus on the special case when the modulus is a prime number. This simplifies certain operations, e.g. we have seen in the previous document that for a prime modulus, every nonzero remainder is invertible:

Theorem 1.2. *If* p *is prime, then all* $a \not\equiv 0 \pmod{p}$ *numbers are invertible modulo* p.

The multiplication also works better if the modulus is prime:

Theorem 1.3. *If* p *is prime,* $a \not\equiv 0 \ (p)$ *and* $b \not\equiv 0 \ (p)$*, then* $a \cdot b \not\equiv 0 \ (p)$.

That is, the product of nonzero remainders never gives a zero remainder. This corresponds to the fact that the product of two nonzero real numbers is never zero. It seems very natural, and it will be very useful, but this is only true if the modulus is prime! For example, if the modulus is $m = 10$, then $4 \cdot 5 \equiv 20 \equiv 0 \pmod{10}$. (Later such numbers will be called *zero divisors*.)

And since the exponentiation is just repeated multiplication, powers of nonzero remainders are also nonzero:

Theorem 1.4. *If* p *is prime and* $a \not\equiv 0 \ (p)$*, then* $\forall n \in \mathbb{N} : a^n \not\equiv 0 \ (p)$.

(But again, this is only true for a prime modulus, e.g. $6^2 \equiv 0 \pmod{12}$.)

In Sage, there are multiple ways to calculate the modular power $a^n \bmod m$. The naive method, `a^n % m`, is not efficient for large exponents, because it first calculates a^n, which can be very big, and then it takes its remainder, i.e. it throws away most of the number. But there is a much more efficient function, `power_mod()`:

```
sage: 5^99999999 % 7              # SLOW!
6
sage: power_mod(5, 99999999, 7)   # fast
6
```

2 Fast exponentiation

How can we calculate the modular power $a^n \bmod m$ efficiently – without using Sage functions? By using an algorithm called **fast exponentiation** (or *exponentiation by squaring*), which works by rearranging the power into a much more efficient form. (It is also used by Sage's `power_mod()`.)

For example, how can we calculate a^8? The naive method is $a^8 = a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$, which is seven multiplications. Can we do better? Yes, we can: $a^8 = \left(a^4\right)^2 = \left(\left(a^2\right)^2\right)^2$, which is only three multiplications (three squarings). This can be generalized as follows: if the exponent is even, then the first step is the same: $a^{2n} = \left(a^n\right)^2$, and we can continue it as long as the inner n remains even; and if it's odd, we can seperate an a: $a^{2n+1} = \left(a^n\right)^2 \cdot a$, and continue in the same way. For example:

$$a^{22} = \left(a^{11}\right)^2 = \left(\left(a^5\right)^2 \cdot a\right)^2 = \left(\left(\left(a^2\right)^2 \cdot a\right)^2 \cdot a\right)^2.$$

In this way, calculating a^{22} requires not 21, but only 6 multiplications.

Notice that the above representation of a^{22} has a strong connection to the binary form of 22. In each step, we perform a squaring, and optionally multiply the result by a. If we write '1' each time this multiplication happens, and a '0' otherwise, and prefix the whole string by a '1' (for the innermost a), then we get the binary representation of the exponent, e.g. $22 = 10110_2$.

Since we do modular exponentiation, we need to take the remainder – but not just at the end, but after each step, otherwise the number could grow very big, making "fast exponentiation" slow.

Fast exponentiation(a, n, m)**:**

Input: $a \in \mathbb{Z}$ (the base), $n \in \mathbb{N}^+$ (the exponent), $m \in \mathbb{N}^+$ (the modulus)
$B :=$ Binary representation(n) (see below)
$r := a := a \bmod m$
for $b :=$ *elements of* B *backwards, except the last* 1 **do**
  $r := r \cdot r \bmod m$
  if $b = 1$ **then**
    $r := r \cdot a \bmod m$
Output: r

Binary representation(n)**:**

Input: $n \in \mathbb{N}^+$
Output: B, the list of binary digits in reverse order: $n = (B[l] \dots B[1]B[0])_2$, i.e.
$n = B[0] + B[1] \cdot 2 + B[2] \cdot 2^2 + \dots + B[l] \cdot 2^l$ (where $B[l] = 1$)
$B :=$ *(empty list)*
while $n \neq 0$ **do**
  add $n \bmod 2$ to the end of B
  $n := n \operatorname{div} 2$

This algorithm is very efficient, because it performs at most $2L$ multiplications, where L is the binary length of the exponent, i.e. $L \approx \log_2 n$. E.g. for $n = 1000000$, this it at most 40 multiplications.

Note: fast exponentation works not only for modular exponentation, but also for exponentiating any kind of mathematical object that supports multiplication (e.g. integer, real number, matrix etc.). Then the remainder calculations (mod m) are obviously omitted.

3 Structure of the powers

Now let's examine the modular powers of various numbers modulo 7:

```
sage: matrix([[power_mod(a, n, 7) for n in range(30)] for a in range(1, 7)])
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1]
[1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6]
[1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1]
[1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6]
[1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6]
```

First, notice that the powers repeat periodically. The repetition is not surprising, because there are only finitely many remainders, so the powers cannot be all different. We can also notice that the first power that repeats is always $1 (= a^0)$.

Why is that so? Because if we have any repetition, e.g. $a^8 \equiv a^2 \ (7)$, then we can multiply it by the modular inverse of a^2, and get an earlier repetition: $a^8(a^2)^{-1} \equiv a^6 \equiv 1 \ (7)$.

But *when* does this repeation happen first?

Definition 3.1. The **order** of $a \in \mathbb{Z}$ modulo m is the smallest exponent $n \in \mathbb{N}^+$ for which $a^n \equiv 1 \pmod{m}$. Notation: $\mathrm{ord}_m(a)$.

Example. $\mathrm{ord}_7(4) = 3$, i.e. the order of 4 modulo 7 is 3, because $4^3 \equiv 1 \ (7)$, but none of the earlier powers are 1: $4^1 \equiv 4 \ (7)$ and $4^2 \equiv 2 \ (7)$. Or, we can also see from the above table that $\mathrm{ord}_7(5) = 6$.

So the order is the smallest positive exponent for which the modular power is 1. But since 1 is the first repeated power, it means that the order is also the length of the period of the repetition: $a^{k+\mathrm{ord}_m(a)} \equiv a^k a^{\mathrm{ord}_m(a)} \equiv a^k \cdot 1 \equiv a^k \pmod{m}$.

We can also see from the table that the sixth power of each nonzero remainder is 1 (but this is not always the first 1). This can be generalized as follows:

Theorem 3.2 (Fermat's little theorem)**.** *If p is prime and $a \not\equiv 0 \pmod{p}$, then $a^{p-1} \equiv 1 \pmod{p}$.*

So the order of any number can be at most $p-1$, i.e. $\mathrm{ord}_p(a) \leq p-1$. But we can say something even more specific about the order:

Theorem 3.3 (Lagrange's theorem)**.** *If p is prime and $a \not\equiv 0 \pmod{p}$, then $\mathrm{ord}_p(a) \mid p-1$.*

Indeed, we can see that the order of the numbers are 1, 2, 3 or 6, which are the divisors of 7–1 = 6.

There are two special cases of note. First, of course $\mathrm{ord}_m(1) = 1$ for any modulus, because any power of 1 is 1. Second, we can see that $\mathrm{ord}_7(6) = 2$, which can be generalized as follows: $\mathrm{ord}_m(m-1) = 2$, because $m-1$ is congruent to -1, and $(-1)^2 = 1$, i.e. $(m-1)^2 \equiv (-1)^2 \equiv 1 \pmod{m}$ (an exception is $m = 2$, where this is not the first 1, because $-1 \equiv 1 \pmod{2}$).

It is also worth noting that the the modular inverse of any number can be found in the list of its powers. Why? Because Fermat's little theorem says that $a^{p-1} \equiv 1 \ (p)$, and multiplying this by a^{-1} gives $a^{p-2} \equiv a^{-1} \ (p)$. The first occurance of a^{-1} is $a^{-1} \equiv a^{\mathrm{ord}_p(a)-1} \ (p)$. (E.g. for $p = 7$, the inverse of 4 is 2, its order is 3, and indeed: $4^{3-1} \equiv 2 \ (7)$.) This gives a new method for finding the modular inverse (in addition to the ones described in the previous document).

Now consider those numbers whose order is the largest possible value:

Definition 3.4. For a prime p, the integer $g \in \mathbb{Z}$ is a **primitive root** modulo p if $\mathrm{ord}_p(g) = p-1$. (It is also called a **generator**.)

Example. 3 and 5 are primitive roots modulo 7, because their order is 6 (see the table from earlier).

Why are primitive roots interesting? For example, the powers of 5 modulo 7 are the following: $1, 5, 4, 6, 2, 3, 1, 5, \ldots$. Notice that this list contains every nonzero remainder. This is true in general:

Theorem 3.5. *If p is prime and g is a primitive root modulo p, then $\forall a \not\equiv 0\ (p) : \exists n \in \mathbb{N} : g^n \equiv a\ (p)$.*

In other words: a primitive root generates every possible nonzero remainder by its powers.

(The proof is simple: the order of the primitive root is by definition $p-1$, so the first repetition in the list happens after $p-1$ powers, which means that the first $p-1$ powers are all different; but there are exactly $p-1$ remainders, so all of them must be present in the list.)

We can ask the question whether primitive roots exist for every possible prime p. The answer is yes (but the proof is difficult, so it is omitted):

Theorem 3.6. *If p is prime, then there exists a primitive root modulo p.*

4 Discrete logarithm

So a primitive root generates all nonzero remainders, and each one exactly once within the first period. So we can ask the following question: where can we find a given remainder in the list? Notice that this question is the inverse of exponentiation, which asks for the value of the power from the base and the exponent; now we ask for the exponent if we know the base and the power. This is very similar to what logarithm does for real numbers: the real equation $b^x = a$ has the solution $x = \log_b a$, e.g. the solution of $2^x = 4\sqrt{2}$ is $x = \log_2 4\sqrt{2} = 2.5$. So we need the modular analogue of logarithm:

Definition 4.1. The **discrete logarithm** (or **index**) of $a \in \mathbb{Z}$ to the base $b \in \mathbb{Z}$ modulo m is the smallest exponent $n \in \mathbb{N}$ for which $b^n \equiv a\ \pmod m$. Notation: $x = \operatorname{ind}_b a\ \pmod m$.

Example. The base-5 discrete logarithm of 6 modulo 7 is 3, i.e. $\operatorname{ind}_5 6 = 3\ (7)$, because $5^3 \equiv 6\ (7)$.

By the above reasoning, if g is a primitive root modulo p, then $\operatorname{ind}_g a$ exists for all $a \not\equiv 0\ (p)$.

But how can we calculate the discrete logarithm? The simplest way is linear search, i.e. to calculate all modular powers of g up to g^{p-2} – by a multiplication and a remainder calculation in each step –, and whichever matches a, its exponent will be the discrete logarithm. But this is very slow for large p, e.g. for $p \approx 1000000$, it needs one million multiplications in the worst case.

But no really efficient method is known; the discrete logarithm problem is still an open question of mathematics today. There are *somewhat* more efficient methods than the above linear search (such as the one described in the next section), but none of them works for such large numbers that fast exponentiation can still easily handle. So we can do modular exponentiation efficiently, but not its inverse, the discrete logarithm.

But this is not a problem. In fact, as we'll see, this is good news!

Sage can calculate the discrete logarithm using the `discrete_log()` function (which is much more general than what we need, so it's a bit verbose to call):

```
sage: discrete_log(mod(6, 7), mod(5, 7))   # base-5 log 6 (mod 7)
3
sage: power_mod(5, 3, 7)    # check
6
```

But of course, if p is very big, `discrete_log()` can't solve it either:

```
sage: p = next_prime(2^1000)
sage: a = power_mod(7, randrange(p), p)
sage: discrete_log(mod(a, p), mod(7, p))    # won't work
...
```

4.1 Baby-step giant-step

A slightly more efficient method for discrete logarithm is the **baby-step giant-step** algorithm. The idea is to take a number M in the middle, i.e. for which $M^2 \approx p-1$, and write the unknown exponent of $a \equiv g^n \ (p)$ in the form $n = iM + j$, where $0 \leq i, j < M$ (e.g. if $M = 10$ and $n = 76$, then $n = 7M + 6$, i.e. $i = 7$ and $j = 6$). First, we calculate the first few g^j powers (modulo p): $1,\ g,\ g^2,\ \ldots,\ g^{M-1}$ ("baby-step"), and store them. Then we start from the power a, and multiply it repeatedly by g^{-M}: $a,\ g^{-M}a,\ g^{-2M}a,\ \ldots$ ("giant-step"). These two sequences (the "baby-step" and the "giant-step") will eventually meet: when a is multiplied by exactly g^{-iM} (where i is from the exponent $n = iM + j$, i.e. it is unknown), then $g^{-iM}a = g^{-iM+n} = g^{-iM+(iM+j)} = g^j$, which is among the stored g^j values. When it happens, we have i and j, and so we can construct n from them.

Baby-step giant-step algorithm:

Input: p prime, $a \in \mathbb{Z}$ (the power), g primitive root modulo p
$M := \left\lceil \sqrt{p-1} \right\rceil$
$b := 1$
for $j = 0, 1, \ldots, M-1$ **do**
  Store (b, j) in a table. (Note: $b \equiv g^j \ (p)$.)
  $b := b \cdot g \bmod p$
$c := b^{-1} \bmod p \ \ (= g^{-M} \bmod p)$
$b := a$
for $i = 0, 1, \ldots, M-1$ **do**
  if *b is in the table for some j* **then**
    the result is: $\rightarrow n := iM + j$
  $b := b \cdot c \bmod p$

In order for this algorithm to be efficient, our "table" must store and find elements very quickly – for example, we can use a hash table. The key of the table is b, and the value is the index j. In Sage, we can use Python's dictionary (`dict`) type for this (empty dict: `T = {}`, insert an element: `T[b] = j`, find an element: `b in T`, access that element: `T[b]`).

The algorithm performs $\sim 2\sqrt{p}$ multiplications, which is much better than the linear search (e.g. for $p \approx 1000000$, it does ~ 2000 multiplications). But if we compare it to fast exponentiation, then it is still slow for large p. Also, it requires much more memory too (it stores $\sim 2\sqrt{p}$ numbers).

5 Diffie–Hellman key exchange

Assume that two people would like to communicate privately through a public channel (e.g. the internet). So they use encryption, but for that, they need a *secret key* that allows them to read each other's encrypted messages. The problem is that they have never met, so they can only communicate the secret key through the same public channel, where anything they say can be overheard by someone else. How can they still establish the shared secret key between each other?

This seemingly impossible problem can be solved by the **Diffie–Hellman key exchange**. This method is based on the fact that the discrete logarithm problem is hard, i.e. if we have a secret exponent $n \in \mathbb{N}$, then we can share the modular power $g^n \bmod p$ publicly (along with g and p), because nobody will be able to calculate the secret exponent n back from them.

The Diffie–Hellman key exchange works as follows (every exponentiation is meant modulo p):

1. The two participants, Alice and Bob agree on a prime number p and a primitive root g modulo p (these values can be shared publicly, or they can even be fixed constants).
2. They both generate their own random secret exponent between 1 and $p-1$: a and b, where a is only known by Alice, and b is only known by Bob.
3. Alice calculates g^a, and sends it to Bob.
4. Bob calculates g^b, and sends it to Alice.
5. Then they both calculate g^{ab} from the data they know:
   - Alice knows a and g^b, so she calculates $g^{ab} = \left(g^b\right)^a$.
   - Bob knows b and g^a, so he calculates $g^{ab} = (g^a)^b$.

So the common secret is g^{ab}, which both Alice and Bob can calculate with the above steps. But anyone else who listens on the channel can see only g, g^a and g^b, from which they cannot calculate g^{ab}, because they also need one of the secret exponents (a or b), and for that, they would need to solve the discrete logarithm problem.

Example.

1. Let $p = 23$ and the primitive root $g = 5$.
2. Alice generates the secret exponent $a = \mathbf{7}$, and Bob generates $b = \mathbf{21}$.
3. Alice calculates $g^a \equiv 5^7 \equiv \mathbf{17} \pmod{23}$, and sends it to Bob.
4. Bob calculates $g^b \equiv 5^{21} \equiv \mathbf{14} \pmod{23}$, and sends it to Alice.
5. Then they both can calculate g^{ab}:
   - Alice: $g^{ab} \equiv \left(g^b\right)^a \equiv 14^7 \equiv \mathbf{19} \pmod{23}$.
   - Bob: $g^{ab} \equiv (g^a)^b \equiv 17^{21} \equiv \mathbf{19} \pmod{23}$.

In order for the Diffie–Hellman key exchange to be secure, the prime p must be large enough (and it must satisfy certain other mathematical requirements, which are not detailed here). In practice, p is several thousand bits long (e.g. 2048 bits long, which means that its binary representation has 2048 digits, which is 617 digits in decimal), see e.g.: [5].

The Diffie–Hellman key exchange is used in various places, including internet systems such as HTTPS, SSH, VPNs etc. – usually in combination with other cryptographic algorithms like RSA and AES (which will be discussed in later documents).

6 Random number generators

Most programming languages have a *random number generator (RNG)*. But most of them don't generate "true" random numbers (which can be created e.g. by throwing dice or by other physical processes), instead they calculate the numbers using a simple algorithm, which produces numbers that look "random enough". Such numbers are called *pseudorandom numbers*.

A very simple pseudorandom number generator (PRNG) can be created using modular exponentiation. Although the powers repeat periodically, they look quite random within one period (e.g. the powers of 5 modulo 7 are: $1, 5, 4, 6, 2, 3$). For example, C++'s `std::minstd_rand` [6] generator uses the following parameters:

$$x_n := 48271^n x_0 \bmod 2147483647,$$

where x_0 is an arbitrary initial value called the *seed* (which is often calculated from the current time, to make it always different); the modulus $p = 2147483647 = 2^{31}-1$ is a prime number that is chosen so that the remainders just fit into a signed 32-bit integer (and that the remainder calculation is efficient); and the base, 48271 is a primitive root modulo p.

This random number generator is very simple and very efficient, so it is well-suited e.g. for simulations and games. However, it is not suitable for generating the exponents of the Diffie–Hellman key exchange, a and b (nor in any other algorithm where security is important), because its simplicity means that it is very easy to predict the next numbers, which makes the whole key exchange useless.

Instead, *cryptographically secure pseudorandom number generators* (CSPRNG) should be used, which are more complex, but they have sufficient mathematical guarantees that an outside listener can't predict the subsequent numbers (nor parts of them, not even with certain probability).

Many programming langauges, including Sage (and Python) have both secure and non-secure random number generators:

```
sage: randrange(100)            # NOT secure
94
sage: import secrets
sage: secrets.randbelow(100)    # secure
67
```

7 Author

This document was written by Uray M. János, partly using the following sources:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa (Hungarian)

[2] https://en.wikipedia.org/wiki/Modular_exponentiation

[3] https://en.wikipedia.org/wiki/Discrete_logarithm

[4] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[5] Primes for the Diffie–Hellman key exchange: https://www.rfc-editor.org/rfc/rfc3526

[6] https://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine.html

Errors and suggestions can be reported here: uray.janos@inf.elte.hu.